RESEARCH PAPER
Contrib. Sci. & Tech Eng, 2024, 1(4)
Homepage: https://cste.journals.umz.ac.ir/
DOI: 10.22080/cste.2025.28329.1009

Intuitive AsyncAPI Modeling: Design and Evaluation of a Purpose-Built Graphical Editor

Leila Samimi-Dehkordi [1]*, Delaram Nikbakht Nasrabadi [2]

[1] *Assistant Professor, Department of Computer Engineering, Faculty of Engineering, Shahrekord University, Shahrekord, Iran.*

[2] *Master Student (if applicable), Department of Computer Engineering, Faculty of Engineering, Shahrekord University, Shahrekord, Iran.*

Article Info
Received 04 January 2025
Accepted 21 January 2025
Available online 07 March 2025

Keywords:
AsyncAPI;
Graphical Modeling Editor;
Usability;
User Experience;
Message-Driven Architecture.

Abstract:
This paper explores the challenges in developing AsyncAPI specifications by presenting the design and evaluation of a dedicated graphical editor. This research aims at enhancing usability and productivity and reducing errors associated with AsyncAPI modeling compared to traditional textual and tree-based approaches. The research method involved the design of a graphical editor integrated into the Eclipse environment using Eclipse Sirius with a model-driven development approach based on the Ecore metamodeling framework. The evaluation was based on four case studies with varying levels of complexity and a questionnaire for 40 participants in which the graphical editor was compared to YAML and tree-based representations regarding its understandability, proneness to errors, and modeling efficiency. The discussion focused on the users' feedback. The results show that the graphical editor significantly improves usability and reduces errors, particularly for complex cases, thus facilitating a faster grasp of component interrelationships and efficient error detection. While the editor was generally positively assessed, some problems related to scalability for large models and the Eclipse-based infrastructure were reported. In summary, this study illustrates the capacity of graphical modeling to revolutionize AsyncAPI development by providing a more intuitive and effective alternative to conventional textual approaches; however, subsequent research must focus on scalability and platform accessibility in order to enhance widespread utilization.

***Corresponding Author:** samimi@sku.ac.ir

Supplementary information: Supplementary information for this article is available at https://cste.journals.umz.ac.ir/

Please cite this paper as: Samimi-Dehkordi, L. , & Nikbakht Nasrabadi, D. (2025). Intuitive AsyncAPI Modeling: Design and Evaluation of a Purpose-Built Graphical Editor. *Contributions of Science and Technology for Engineering*, 1(4), 44-57. doi: 10.22080/cste.2025.28329.1009.

1. Introduction

Asynchronous, message-driven architectures are one of the cornerstones of most modern distributed systems, enabling communication between scales and resilient software components. In these systems, the key to finding a good message-passing in throughput and latency is to implement, set up, and control a communication interface [1]. It provides developers with a standard way to write and document these interfaces and empowers the community with the tools necessary to define event-driven interoperability with ease. As AsyncAPI became more prevalent in micro-services architecture, developers needed tools to create, edit, and manage specifications [2].

Although the textual syntax of AsyncAPI can be expressive and powerful, developers may find it challenging to understand the specification's intricacy [3]. To tackle this issue, a recent study investigated the use of graphical editors to design AsyncAPI definitions [4]. These types of editors help limit mistakes, improve the user experience, and make the big-picture flow of events around a graphical representation of the underlying data model much more efficient [5].

The paper on "model-driven development of asynchronous message-driven architectures with AsyncAPI" is one of several noteworthy contributions in this area [4]. This research discussed the pros and cons of graphical syntax and textual syntax to represent AsyncAPI specs. It emphasized some advantages of graphical editors such as minimizing errors and intuitive modeling. It also mentioned some challenges with adapting AsyncAPI-specific aspects using Ecore annotations. The flexibility offered by this approach was valued by those familiar with metamodeling, but it was challenging and less available to those who were not. The aforementioned study's limitations were associated with Ecore annotations rather than graphical tools. As an illustration, participants expressed that manually annotating models was cumbersome. As the model got bigger, the effort required to connect their annotations to specific sections of the code became too much to handle. While the graphical editors were recognized for their abstractions and ease of use, it was balanced out by an annotation-based approach

ISSN ***3060-6578***

that limited casual modelers from being able to access and efficiently work with the models. Results like these validate the need for a more intuitive and sophisticated graphical tool that removes the "barrier" while retaining much of the power of visual modeling.

Thus, the present work aims to create and validate a new tool that graphically aids in the specification of AsyncAPI. By removing the need for manual annotations and providing a more uniform and effective modeling experience, the suggested editor overcomes the drawbacks of earlier methods. It guarantees compatibility with current development environments, offers streamlined editing processes, and enables integrated management of AsyncAPI-specific data. By reducing complexity and improving usability, this editor will help developers with different skill levels intuitively create and maintain AsyncAPI specifications.

The purpose of this study is to address the following research questions:

RQ1: Is the graphical editor more usable, and does it lower the learning curve for developers, compared to tree-based representations and YAML?

RQ2: Does the graphical editor reduce errors compared to both tree-based and YAML approaches for AsyncAPI modeling?

RQ3: How do various representations, including graphical, tree-based, and YAML-based formats, compare in terms of efficiency and user experience in AsyncAPI modeling?

The study intends to analyze the new editor's practical sense concerning the AsyncAPI modeling process through these research questions. In addition to eliminating tedious tasks to make modeling more interactive and efficient, all of those features that users can interact with at a high level are put in the editor.

The rest of the paper is structured as follows: Section 2 introduces a motivating example. Section 3 provides a detailed literature review of the AsyncAPI tooling and graphical modeling. Section 4 describes the design/development process that led to the proposed graphical editor, along with important features and architectural choices. In section 5 we present the evaluation methodology as well as user studies and metrics used to evaluate the editor with respect to existing methodologies. Section 6 details and offers the results of the analysis, while Section 7 discusses the implications for the research findings and potential limitations of the approach. Section 8 provides the conclusion, summarizing our contributions and directions for future work.

2. Motivating Example

Asynchronous communication patterns are used in designing scalable and robust software systems as the world grows more interconnected. Asynchronous messaging frameworks are becoming essential to contemporary software development, from event-driven architectures in micro-services to Internet of Things platforms that process data streams in real-time [6]. AsyncAPI is one of the specifications that has emerged to describe and document such systems in a more structured format, specifying how messages are exchanged, which channels are used, and the event-driven business processes [2]. However, like with most domain-specific languages, AsyncAPI is a text-based language where the API contract is defined in YAML or JSON format and hence can be overwhelming for developers who might not be familiar with these syntaxes/complex architectures [4]. Now to understand how AsyncAPI would be, let us consider the IoT system with temperature and humidity sensors that communicate asynchronously with the central servers (Figure 1): There are two sensors: a temperature sensor connected over a channel and reports its integer temperature in degrees Celsius, and another would be a humidity sensor that would report its percentage in number format.

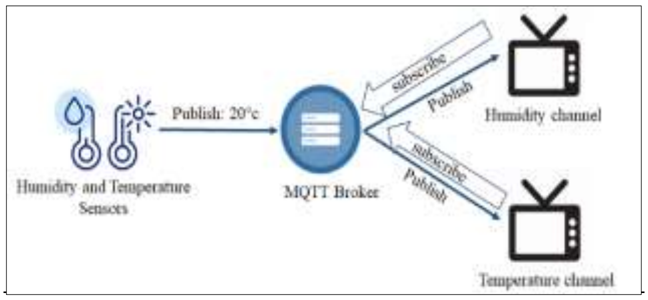
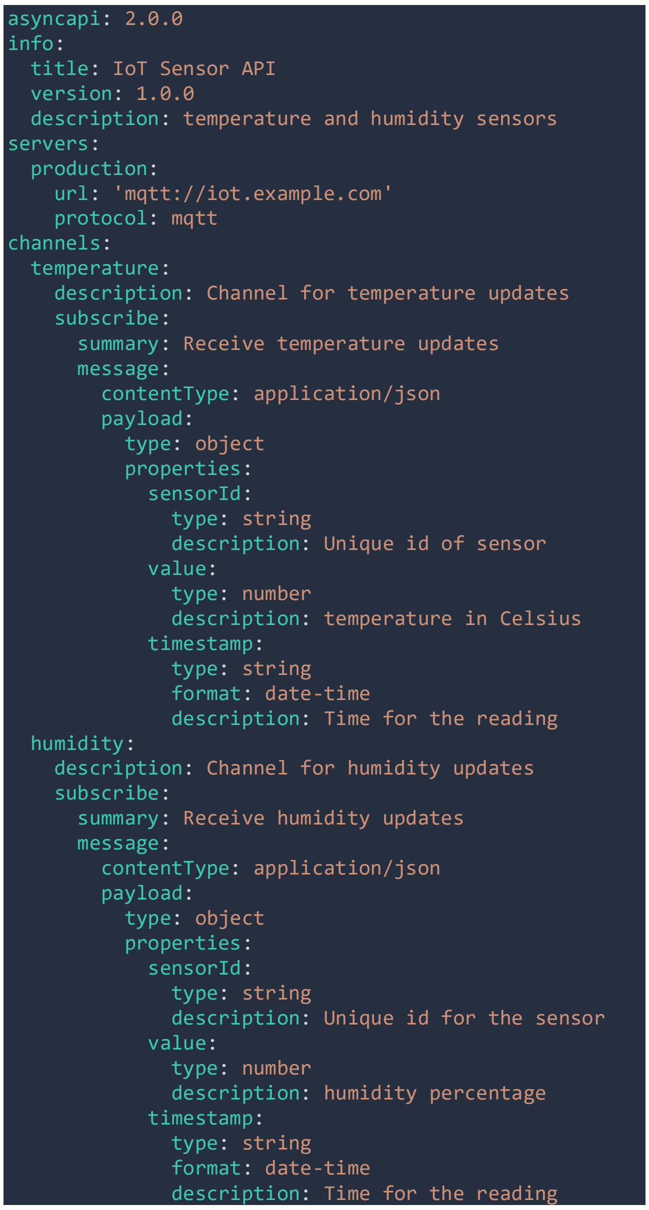


Figure 1. An IoT sensor data flow using the MQTT protocol, as defined by an AsyncAPI specification

Figure 2 shows this system is documented using AsyncAPI. In the example, the AsyncAPI code defines an API for an IoT sensor system that publishes temperature and humidity data. It supports asynchronous communication over the MQTT protocol. It creates two channels in the channels part: one for temperature updates and the other for humidity updates. Every communication channel has a message object that describes what can be published on that channel. The attributes *sensorId*, value, and timestamp are all representative of important sensor information. The servers section defines the MQTT broker serving those messages. With AsyncAPI you may be losing detail on the contract guaranteeing interoperability between the sensor and the subscribers. Basically, at its core, this is some code that describes the data flowing out of the IoT sensor in a machine-readable manner, which ultimately makes it simpler to build applications based on this data for developers.

Once the AsyncAPI specifications for a system are established, they can be reused as part of any of the following steps of the system development and deployment life cycle: 1) server and client implementation, 2) code generation, 3) testing and validation, 4) integration with other systems.

The description is quite complex and contains a lot of details, but it is somewhat complicated in case users are not familiar with structured formats and the system could also get complicated as it grows. Moreover, adopting such representations typically involves a long-term development cycle and a steep learning curve new users have to go through.



```yaml
asyncapi: 2.0.0
info:
  title: IoT Sensor API
  version: 1.0.0
  description: temperature and humidity sensors
servers:
  production:
    url: 'mqtt://iot.example.com'
    protocol: mqtt
channels:
  temperature:
    description: Channel for temperature updates
    subscribe:
      summary: Receive temperature updates
      message:
        contentType: application/json
        payload:
          type: object
          properties:
            sensorId:
              type: string
              description: Unique id of sensor
            value:
              type: number
              description: temperature in Celsius
            timestamp:
              type: string
              format: date-time
              description: Time for the reading
  humidity:
    description: Channel for humidity updates
    subscribe:
      summary: Receive humidity updates
      message:
        contentType: application/json
        payload:
          type: object
          properties:
            sensorId:
              type: string
              description: Unique id for the sensor
            value:
              type: number
              description: humidity percentage
            timestamp:
              type: string
              format: date-time
              description: Time for the reading
```

Figure 2. The AsyncAPI specification for IoT sensors

To address the mentioned limitations, we present the model-based approach proposed in the current paper for the representation of AsyncAPI specifications. In a sense, this creates visual models, allows graphical editing of inappropriate editors, and makes the interface much more user-friendly, intuitive, and error-reducing. This strategy not only encourages more intuition but also enables better collaboration among groups and the efficient refinement of the iterative pattern in complex systems.

3. Related Work

Tools and approaches that make asynchronous API development and management easier have gained a lot of attention due to the growing usage of asynchronous message-driven architectures [7]. AsyncAPI has emerged as a highly intriguing standard for defining and documenting event-driven communication in this context [8]. AsyncAPI gives developers a rich framework to describe message exchanges as it is specifically made for asynchronous interactions, even though it is based on OpenAPI principles [9]. AsyncAPI specifications offer benefits, however, they are also basically complicated, which causes problems, particularly for inexperienced developers who use its textual syntax [4].

One of the primary subjects of discussion in the AsyncAPI community is lowering cognitive load and the possibility of errors when reading textual representations. Although text editors offer greater flexibility and specificity, they necessitate a thorough understanding of the AsyncAPI's syntax and structure [10]. There are tools, like the AsyncAPI Generator and AsyncAPI Studio, which aim to help developers create and validate their specifications more easily. Most of them, however, rely on textual input and can be quite error-prone and tedious for anyone not familiar with the intricacies of AsyncAPI.

To overcome the limitations of textual tools, researchers have proposed graphical editors as a complementary approach [11]. Graphical editors use visual modeling techniques for the representation of API specifications, providing developers with the workspace of abstract representation of message flows, schemas, and bindings [12]. This reduces the risk of syntax errors by providing stakeholders from various engineering backgrounds using a more abstract understanding of the system, facilitating better communication and comprehension.

In this regard, model-driven development of asynchronous message-driven architectures with AsyncAPI is a significant step forward. The seminal work in this field is called "Model-driven development of asynchronous message-driven architectures with AsyncAPI" [4]. It demonstrated an editor based on Ecore annotations and conducted user research to validate it. According to the study participants, the graphical syntax is a more advantageous and visual method of creating AsyncAPI definitions, particularly when bootstrapping a working specification. The editor's utility could be increased by simply integrating it with the Eclipse development environment.

However, the study by Gómez et al. [4] also found some limitations of the Ecore-based editor. A critical issue was that it relies on metamodel annotations to provide AsyncAPI-specific information. This made it very powerful for advanced users, but it posed a significant hurdle for most users with limited metamodeling knowledge. The connection between annotations and code generation was not always clear, requiring users to expend additional effort in figuring out the impact of their models. The second difficulty that was reported in this category was that the graphical modeling is typically point-and-click, which is slow and prone to error: "Creating and editing specifications was tedious and sometimes also for complex systems [in sets of graphical modelers]. This issue means that graphical editors need to start evolving to contain better workflows and more tools that help repeat tasks. Furthermore, even though the specification can be discussed at a more abstract level using the graphical view, it is not always possible to understand its details without switching between the textual and graphical representations.

In parallel, model-driven development (MDD) has proven to be a partly addressed area for a graphical tool for studies of system complexity [13]. MDD focuses on higher-level abstractions with automation to accelerate development processes. As a sort of comparison, tools like UML-based editors or domain-specific modeling languages (DSMLs) have been successful; a new sort of model is being developed, only not to do with code [14, 15]. While these principles are applicable in the context of AsyncAPI, they need to be adjusted about certain features of AsyncAPI that are related to schema validation, protocol bindings, and code generation [3, 4, 16].

Effective design techniques are revealed by the use of graphic editing tools in various domains, such as SysML (Systems Modeling Language) and BPMN (Business Process Model and Notation) [5, 17]. These tools are designed with usability, modularity, and integration into existing toolchains, all of which are also aspects of AsyncAPI tooling. However, the unique difficulties of asynchronous APIs—like simulating dynamic message flows and event-based communication—emphasize the necessity for customized solutions.

Table 1 compares various approaches in the context of graphical editing capabilities, AsyncAPI support, and code generation. Our method stands out by focusing on the design of a graphical editor tailored for AsyncAPI specifications. This approach leverages the code generation engine outlined in the original paper to automate the creation and implementation of message-driven infrastructures. By integrating graphical modeling with AsyncAPI compliance, our method enhances usability while streamlining the development process for asynchronous architectures.

Table 1. Comparison of related work

Source	Graphical Editing	AsyncAPI Support	Code Generation
Arslan et al. [14]	Discusses graphical modeling for IoT-based transportation.	Not specified.	Not specified.
Bedini et al. [18]	Proposes graphical editor creation with Eclipse Sirius.	Not specified.	Focuses on editor generation.
Budinsky [19]	Introduces EMF supporting graphical modeling via GMF.	Not specified.	Provides extensive code generation.
David et al. [15]	Analyzes tools with graphical editing capabilities.	Not specified.	Discusses code generation across tools.
Gómez et al. [4]	Proposes editors for asynchronous architectures.	Supports AsyncAPI (unspecified versions).	Automates message-driven design.
Lercher et al. [8]	Discusses micro service API evolution, no focus on graphics.	Not specified.	Not specified.
Ordoñez et al. [13]	Reviews MDE approaches with graphical modeling.	Not specified.	Discusses accessibility-related code generation.
Oriol et al. [3]	Not specified.	Focuses on AsyncSLA for agreements.	Not specified.
Rabii et al. [5]	Explores prototyping graphical editors for DSLs.	Not specified.	Discusses prototyping methods.
Ray [12]	Highlights visual programming for IoT applications.	Not specified.	Not specified.
Silva [2]	Not specified.	Suggests AsyncAPI-first design.	May improve developer experience.
Ternes et al. [11]	Analyzes UI design in modeling tools.	Not specified.	Not specified.
Tzavaras et al. [9]	Not specified.	Discusses OpenAPI for the Web of Things.	Not specified.
Verbruggen and Snoeck [17]	Reviews graphical modeling in practitioner experiences.	Not specified.	Discusses MDE code generation.
Wang et al. [16]	Not specified.	Explores AsyncAPI extensions.	Not specified.
Zafar et al. [10]	Reviews trends in MDE graphical modeling.	Not specified.	Discusses domain-specific code generation.

4. Design and Development of the Graphical Editor

Motivated by providing a user-friendly, efficient, and precise tool for modeling asynchronous, message-driven architectures, the proposed graphical editor for AsyncAPI specifications has been developed. Built on top of the Eclipse ecosystem using Eclipse Sirius [18], it follows a model-driven development (MDD) approach based on the Ecore metamodeling framework [19]. Such a choice allows for interoperability with existing well-known tools, standards, and workflows yet still allows for customization to handle efficiently some specific issues related to AsyncAPI.

4.1. Architecture Overview

The editor's architecture is founded on three fundamental elements: 1) Ecore-based Metamodel, 2) Graphical Interface, and 3) Code Generation Module. The underlying Ecore metamodel defines the structure of AsyncAPI elements. It includes critical abstractions for components. The graphical interface is designed with Eclipse Sirius, this component provides an intuitive drag-and-drop environment to create, edit, and visualize AsyncAPI elements, including channels, messages, schemas, and event flows. It allows users to visually represent connections, like the bindings of messages to channels or associations of schemas. The editor uses Model-to-Text (M2T) transformations to generate valid AsyncAPI definitions from graphical models. This transformation guarantees a consistent relation between the visual model and generated artifacts, reducing manual work and easing syntax errors.

We design the graphical interface for the metamodel of the AsyncAPI in Gómez et al. [4].

Figure 3 illustrates a process of generating code based on an AsyncAPI specification. The AsyncAPI metamodel serves as a template for defining the structure of an AsyncAPI specification. A definition of M2T (Model to Text), a set of rules for transforming a model into programming code, is applied to the metamodel and an instance of an AsyncAPI model. The result of this transformation step is Java programming code. An AsyncAPI model can be represented in different forms, such as a tree-structured format, a graphical representation, and a text-based format in YAML. The process is thus enabled for the automatic generation of code from high-level specifications. It enhances reusability and decreases the development time.

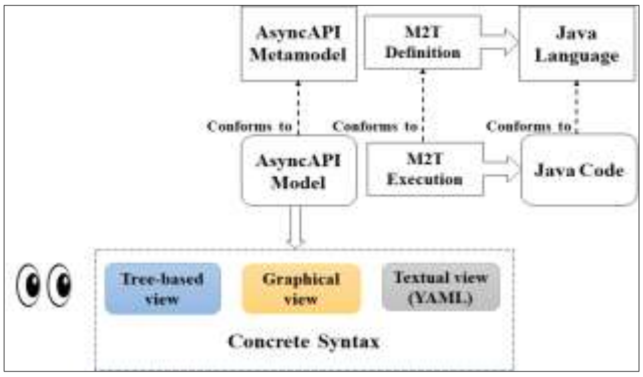


Figure 3. **Process of generating code based on the AsyncAPI specification**

The graphical editor significantly enhances AsyncAPI modeling by adding several important features and innovations. First, it reduces errors through visual validation and implicit rule enforcement. Second, it eases the management of annotations by abstracting away complex Ecore annotations into friendly dialogs. Third, the framework exhibits various levels of abstraction, facilitating the visualization of the overall system architecture and the detailed interactions among components. Furthermore, its seamless integration within the Eclipse ecosystem enhances the development experience. Last, the emphasis on productivity is evident through its drag-and-drop interface and automation of tasks, effectively addressing the challenges typically associated with traditional modeling tools.

The graphical interface of the editor is built on top of Eclipse Sirius, which provides a rich framework for building domain-specific graphical editors. The following key features of Sirius are employed:

- Metamodel-based Design: Sirius uses the Ecore metamodel to define AsyncAPI concepts.
- Customizable Diagrams: Users can design tailored diagrams to visualize AsyncAPI structures.
- Real-time Validation: Sirius ensures immediate feedback on the model's integrity.

The adoption of Sirius not only accelerates the editor's development but also ensures extensibility and scalability for future enhancements.

In summary, the proposed graphical editor uses Eclipse Sirius in combination with model-driven development methods to enable the creation of AsyncAPI specifications. The new features described, are targeting the reduction of errors and enhancement of user experience and productivity, making it one of the most important tools for developers working on asynchronous communication protocols. Integration with a robust Ecore-based metamodel, automated code generation, and intuitive graphical design represents a significant step within the AsyncAPI ecosystem. Table 2 summarizes key AsyncAPI concepts along with concise descriptions and graphical notations.

Table 2. **The key concepts of AsyncAPI**

	Description	Notation
AsyncAPI	The root document object of an API definition combines resource listing and API declaration into a single document. It includes fields like asyncAPI (specification version), info, servers, channels, and components.	-
Info	Provides API metadata, including title, version, description, terms of service, contact information, and license details.	
Servers	A map of Server objects, each representing a message broker or similar program, detailing URLs, protocols, security configurations, and supporting variable substitution. For each server, a message broker, capturing details such as URL, protocol (e.g., HTTP, MQTT, Kafka), protocol version, description, and variables are defined.	
Channel	A map holding relative path names and individual Channel Item objects, representing topics, routing keys, event types, or paths, depending on the protocol or technology used. Each item describes the operations available on a single channel, including fields like description, subscribe (an Operation object), publish (an Operation object), and parameters.	
Operation	Describes a publish or subscribe operation, documenting how and why messages are sent and received, with fields like operation ID, summary, description, and message.	
Message	Describes a message received on a given channel and operation, specifying fields such as name, title, summary, description, and payload (which can be of any type but defaults to a Schema object).	
Schema	Defines input and output data types, including objects, primitives, and arrays. It is a superset of the JSON Schema Specification Draft 7, with fields like title, type, enum, properties, maxItems, minItems, and items.	
Reference	A simple object allowing referencing other components in the specification, both internally and externally, containing only the $ref field, which is a URI.	
Components	Holds a set of reusable objects for different aspects of the AsyncAPI definition, such as schemas, messages, parameters, operationTraits, and messageTraits, which can be referenced using a Reference object.	

These concepts collectively define the structure and elements of an AsyncAPI document, facilitating the description and documentation of asynchronous APIs.

Figure 4 appears to be a screenshot of the designed graphical editor being used to create an AsyncAPI specification. The key components and their roles are as follows.

1. Palette: (on the right side) It contains a variety of elements that can be dragged and dropped onto the diagram to create the AsyncAPI specification. These elements include:
   - Info: For basic information about the API, such as title, version, and contact information.
   - Server: To define the server URL and protocol.
   - Channel: To define communication channels, such as MQTT topics.
   - Operation: To specify the actions that can be performed on a channel (publish, subscribe, etc.).

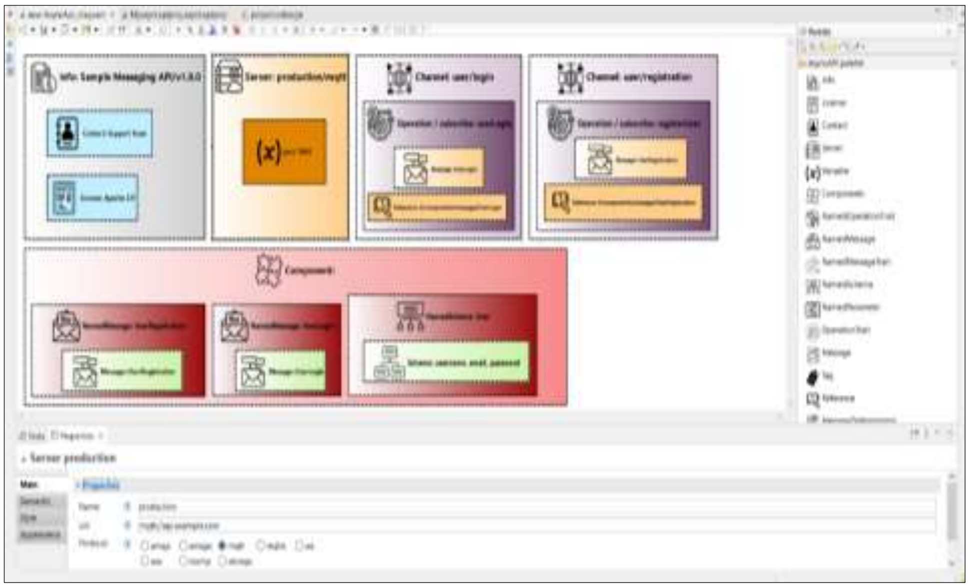


Figure 4. A snapshot of the designed graphical editor

   - Message: To define the structure of messages exchanged over channels.
   - Schema: To define the data structure of messages.
   - Parameter: To define parameters for operations.
   - Component: To define reusable components.
   - Operation Trait: To define traits that can be applied to operations.
   - Message Trait: To define traits that can be applied to messages.
2. Diagram: This is the main area where the AsyncAPI specification is visually constructed. Elements from the palette are dragged and dropped onto the diagram to create the desired structure. The diagram provides a clear visual representation of the API's components and their relationships.
3. Properties View: This view displays the properties of the selected element on the diagram. It allows users to modify the properties of elements, such as the name, description, and data type.

The graphical editor helps developers create and visualize AsyncAPI specifications in a user-friendly way. It provides a visual representation of the API's structure, making it easier to understand and maintain. By using a drag-and-drop interface and a palette of predefined elements, the editor streamlines the process of creating AsyncAPI specifications.

5. EVALUATION

5.1. Evaluation Methodology

To evaluate the applicability and usability of the proposed graphical editor for AsyncAPI modeling, we have developed a methodology based on implementing real-world case studies and collecting participant feedback through questionnaires and interviews. Unlike purely conceptual evaluations, our approach emphasizes practical implementation and user-centered assessment to gain deeper insights into the editor's effectiveness.

The testing activity will begin by creating four real case studies—each representing increasing orders of complexity. Scenarios that will be addressed as part of these case studies include single-channel message flows, multi-channel event sources, complex message schemas, and multiple bindings. Each case study will be implemented through the use of the proposed graphical editor to exhibit features and characteristics in actual scenarios. By emphasizing practical applications, we ensure that the results are pragmatic and represent the potential difficulties that developers may face during real-world implementation.

To test usability, we will recruit participants with a background in computer science to assess the clarity, understandability, and friendliness of the outputs produced by the editor. This will be performed with the help of structured questionnaires supported by follow-up interviews. Participants will examine graphical models, tree-structured hierarchical representations, and YAML code for the same case studies. They will be able to give their assessments based on the set criteria, which consist of clarity, understanding, capability of error identification, and perceived effectiveness. Through a Likert scale, this will allow quantitative assessment, while qualitative data can also be obtained through open-ended questions and interviews that explain in-depth their difficulties and preferences.

Mixing the scenarios modeled in real life with the user-centered feedback approach ensures a complete evaluation. The results shall help understand the applicability and usability of the editor; they will guide further improvement and refinement. That would also open the road for deeper evaluations in successive studies using direct hands-on usage and task-based assessment.

5.2. Case Studies for Study Applicability

The case studies are selected from the website of the AsyncAPI studio tool[*]. As shown in Table 3, the four case studies were selected to showcase the versatility and applicability of our proposed graphical editor across a diverse range of asynchronous communication scenarios and protocols. They represent varying levels of complexity in terms of message structures, communication patterns (e.g., publish/subscribe, request/reply), and underlying protocols (WebSocket, MQTT, AMQP).

Case 1: Real-Time Financial Data Exchange

The first case concerns the API, using WebSocket technology to support real-time financial data exchange with regard to currency trading. In this system, two-way communication between the client and server is effectively achieved through the use of the WebSocket protocol, enabling updates in real-time about market information. It supports a variety of message formats, system status notifications, handling of subscriptions, and even heartbeat signals, hence this ensures continuous and reliable connectivity. It has an efficient architecture with a message transmission in a flat configuration for better fast parsing and lower complexity. The framework is optimized with low latency and high throughput in mind for use cases demanding fast access to dynamic financial information, such as trading systems or market surveillance applications.

Case 2: IoT-Based Smart Streetlight Management

The second case explores the intelligent infrastructure application using an MQTT-based API for the management and monitoring of IoT-enabled streetlight systems. This API is tailored to operate efficiently under resource-constrained environments, hence suitable for low-bandwidth networks that prevail in most IoT applications. It supports functionalities such as remote start/stop of the streetlights, dimming, and real-time monitoring of ambient parameters such as light intensity. API key-based authentication and OAuth2 authorization based on hierarchical topics, schema-based nested payload structures assure well-defined and interpretable data exchanges. The implemented security mechanisms ensure that the communication is secure. Each local government faces such a big challenge in the tasks of centralized control and observation of the distributed systems for efficient operation, so this designed API is very suitable to be used in smart city initiatives.

Case 3: User Registration and Authentication Messaging API

The third case is a messaging API, designed for user registration and authentication services, using the MQTT protocol to ensure effective message delivery. The API provides a lightweight event-driven architecture in which events about users—like registrations and logins—are handled by special channels. Specifically, it defines two main channels: one for user registration events and another for managing login operations. These channels allow subscribed clients to see and analyze associated events almost in real time. The structure of the message payload is defined via a schema and specified in JSON format, which ensures the consistency and clarity of the data being transmitted. The registration channel's payload consists of the following user attributes: username, email, and password. In the same manner, the login channel logs the username and password used for the authentication. With the use of the MQTT protocol, this API has been optimized for the most efficient, reliable, and scalable communication; it is particularly suitable for event-driven architectures, such as notification systems, account management applications, and user observation activities. Moreover, this robustness is further increased with the integration of schema validation, making data interchange between producer and consumer services seamless.

Case 4: IoTBox Monitoring API

The IoTBox Monitoring API is designed to facilitate the monitoring and control of IoTBox devices by providing real-time status updates. Defines the channels used by publish and subscribe to an IoTBox device's operating state: online, offline, maintenance, or even error. The API makes use of AMQP for the communication between clients and devices where clients can connect to IoTBox devices identified by their unique IDs. Key features include monitoring the status of devices, publishing performance updates, and in particular, subscribing to real-time status changes. This API is constructed on its path toward flexibility and scalability through the following multiple constituent parts: operation traits in the standardization of behavior—monitoring or resetting IoTBox devices. In message traits, a set of reusable properties is defined. In schemas, one for the device status update; one for custom messages including a timestamp and status; the structure of the status header. Named parameters and schemas, therefore, guarantee the consistency of the system and allow configurations for a device.

Table 3. **Comparison of the case studies**

Feature	Case 1	Case 2	Case 3	Case 4

[*] https://studio.asyncapi.com/

Protocol	WebSocket (wss)	MQTT	MQTT	AMQP
Purpose	Real-time market updates	Smart streetlight management	User registration and login events	IoT device monitoring and control
Data Structure	Flat payloads	Nested, schema-based payloads	Schema-defined for user attributes	Nested, schema-based payloads
Event Handling	Basic events	Extensive (e.g., dim, turn on/off)	Dedicated to registration/login events	Status updates (e.g., online/offline)
Security	None	API key, OAuth2, OpenID Connect	None	None
Target Audience	Financial applications	IoT/smart city systems	User-based applications and workflows	IoT developers and large-scale solutions
Complexity	Low	High due to hierarchical topics	Moderate	Moderate

Among these four, Kraken WebSockets API is the simplest one designed for basic functionality, which updates financial data in real time. The structure of this API is very flat, and there are not many message types; most of them are repetitive messages, such as ping and pong messages. Thus, it is well-structured and easy to work with. Contrasting it, the Streetlights MQTT API is much more complicated and provides a lot of functionality, such as controlling, monitoring, and dimming. It uses nested payloads, hierarchical topics, and schema-based data definition-enriching capability, but at the same time, it increases the difficulty in its maintenance. Security mechanisms such as API keys and OAuth2 further contribute to its overall complexity.

5.3. Questionnaire for Study of Usability

A structured questionnaire (Table 4) was designed and directed to developers and modelers to discover how different representations (graphical, tree-based, and code-based) are usable, efficient, and error-prone when performing AsyncAPI modeling. This questionnaire was constructed to respond to three research questions: whether the graphical editor increases usability and reduces the learning curve, whether it minimizes errors concerning tree-based and YAML representations, and what the comparison of representations is according to the efficiency of users and their subjective experiences.

Table 4. Questionnaire for the usability of the graphical editor

#	Question	Possible answers
1	Education Level	
2	Have you ever come across AsyncAPI?	Yes/No
3	How familiar are you with asynchronous communication?	1 to 5
4	How familiar are you with YAML?	1 to 5
5	How familiar are you with JSON?	1 to 5
6	How familiar are you with the Ecore modeling tool?	1 to 5
7	How familiar are you with Graphical Editors?	1 to 5
8	How would you rate your understanding of the code?	1 to 5
9	How would you rate your understanding of the tree model?	1 to 5
10	How would you rate your understanding of the graphical model?	1 to 5
11	How long do you estimate it would take you to understand the graphical model?	Number of minutes
12	How long do you estimate it would take you to understand the tree model?	Number of minutes
13	How long do you estimate it would take you to understand the code?	Number of minutes
14	Is a graphical model is created faster than a tree model or generated code?	Yes/No/Not sure
15	Which did you understand the fastest?	graphical/tree-based/code views
16	Which method is less prone to errors?	graphical model/tree model /code
17	Which of the following methods do you find easier?	Drag and drop/ Point-and-click/ Coding all commands and data
18	Which environment is more stable? Stability: The different elements of the user interface should be used in a uniform manner.	graphical editor/ tree-based editor/ code editor
19	How suitable is the design of our graphical editor?	1 to 5
20	If the graphical model editor changes, what would be preferred?	reducing complexity/increasing clarity/ adding new features/no changes
21	If you were to design an AsyncAPI project, which method would you prefer?	graphical editor/ tree-based editor/ code editor/ hybrid method
22	Please explain the reason for your choice in a real AsyncAPI project.	open answer

Subjects were exposed to existing AsyncAPI models, represented as graphical diagrams, tree-based structures, and code snippets. The latter asked them to predict the time needed to work out each representation, rate how well they understood it, and state their preference for creating models based on it and its use in error-prone situations. Other questions assessed participants' prior exposure to asynchronous communication, YAML, JSON, or other modeling tools, at least to put their answers into context for further analysis.

The data reported were analyzed with respect to usability in terms of learning curve, comprehensibility, time needed to understand, proneness to errors in terms of user's perceived reliability, and efficiency, expressed as time required to understand each representation according to ease. That multi-faceted approach could provide comprehensive insights about each modeling method's perception with its developers.

To systematically address the research questions, the questionnaire was designed so that specific questions in the table directly map to each research question (RQ):

RQ1: Questions 7, 8, and 9 measure participants' comprehension of the graphical, tree-based, and code representations. Questions 10, 11, and 12 track the estimated time it would take for someone to get up to speed with each representation; these tell us how steep or flat the learning curve will be. Additionally, Questions 16, 17, 18, 20, and 21 explore participants' preferences, ease of use, perceived consistency, and the appropriateness of the graphical editor. Question 15 evaluates which representation was understood the fastest, further contributing to the understanding of usability.

RQ2: Question 16 directly asks participants to identify which representation they perceive as less prone to errors. The responses reveal user opinions about the reliability and likelihood of making mistakes in each method.

RQ3: Questions 10, 11, and 12 provide data on the time efficiency of each representation by asking participants to estimate the time taken to understand the models. Question 13 identifies which representation was understood the fastest. Additionally, Questions 7, 8, 9, 15, 16, and 17 contribute to comparing user experiences, including comprehension, ease of use, and speed of creation across methods.

In this way, every question in the questionnaire corresponds with each of the research questions; thus, the responses that would be gathered through questionnaires could be analyzed for substantial answers to the objectives of this study. This combination of comprehension ratings, time estimates, and user preferences enables us to comprehensively assess usability, error-proneness, and efficiency.

The study involved 40 participants with diverse educational backgrounds. The majority of participants were either Master's students (30%) or held a Master's degree (25%), followed by PhD holders (20%), Bachelor's degree holders (12.5%), and PhD students (12.5%). Regarding prior experience with AsyncAPI, a significant majority (70%) reported having no previous exposure, while 30% indicated they had encountered it before. Participants' self-assessed familiarity with asynchronous communication was predominantly moderate (45%), with smaller proportions reporting low (20%), very low (15%), high (15%), and very high (5%) familiarity. Familiarity with YAML was generally lower, with 40% reporting very low familiarity, followed by low and moderate familiarity at 27.5% each, and only 5% reporting very high familiarity. In contrast, familiarity with JSON was higher, with moderate familiarity being the most common (37.5%), followed by high (25%), very high (17.5%), low (12.5%), and very low (7.5%). Regarding familiarity with the Ecore modeling tool and graphical editors, the distribution was more balanced. For Ecore, moderate familiarity was reported by 22.5% of participants, with high and very high familiarity each reported by 27.5% and 22.5% respectively, and low and very low familiarity by 10% and 17.5% respectively. For graphical editors, moderate familiarity was again most prevalent (40%), followed by high (32.5%), very high (17.5%), low (7.5%), and very low (2.5%).

6. Results and Analysis

The results of our evaluation focus on assessing the applicability and usability of the graphical editor for modeling AsyncAPI specifications. This section presents findings from our analysis, which compared participant feedback on graphical models, tree-based hierarchical views, and YAML code representations, alongside insights from the case studies modeled using the editor. The results highlight the strengths and areas for improvement in the proposed approach, shedding light on its practical implications.

6.1. Analysis of Applicability

The assessment starts with the application of the graphical editor for modeling AsyncAPI specifications using four different case studies. Each case is studied based on its complexity and the graphical diagram generated using the editor. This section includes diagrams that further illustrate the editor's ability to convey statements at various levels of complexity while maintaining clarity and efficiency.

Case 1: Kraken WebSockets API

It is a relatively simple API, and that translates into a lower complexity for the message definition, as shown in Figure 5. The graphical editor captures the flow and the relation between components very easily, making it easier to understand the complete system quickly. In this particular case, the estimated modeling time was approximately 5 minutes, indicating the editor's appropriateness for straightforward scenarios.

Case 2: Streetlights MQTT API

With more features and nested data structures, this API adds complexity. Although participants pointed out that more features like zooming or filtering would improve usability in these situations, the editor in Figure 6 successfully visualized these elements while preserving clarity. It took about 15 minutes to complete the modeling process, demonstrating the editor's ability to handle moderate complexity with ease.

Case 3: User Registration and Authentication

With fewer nested structures and an emphasis on user-centric operations, this case demonstrates comparatively low complexity. By providing a concise synopsis of the elements and their interactions, the editor facilitated the modeling process. The estimated modeling time was approximately 8 minutes, and participants valued the visual feedback it provided. The graphical model is shown in Figure 4.

Case 4: IoTBox API

The API of the IoTBox (Figure 7) exposes a relatively complex system with a fair amount of features and nested data. The graphical editor made this complexity manageable and easy to see while also simplifying the identification of component dependencies. In this case, the modeling time was around 12 minutes, proving the editor is capable of dealing with mid-level complexities effectively. All of these case studies highlight the editor's adaptability in handling a large variety of situations. Moreover, tree-based models are generated in parallel for all graphical diagrams, meaning the user has the choice of hierarchical views. Furthermore, you can open a tree-based model in a graphical editor, and it automatically creates its graphical version—indicating how seamless the application is between these two representations.

6.2. Analysis of Usability

The evaluation of the graphical editor, informed by participant responses to the questionnaire, provides valuable insights into its perceived strengths and areas for improvement. The survey included 40 participants who responded to a questionnaire to assess their experiences and perceptions of different AsyncAPI modeling approaches, focusing on graphical, tree-based, and code-based methods. Table 5 demonstrates the results. The participants had diverse educational backgrounds, with the majority holding a Master's degree (30%) or a PhD (25%), indicating a relatively well-educated group with potential expertise in the field.

Despite this expertise, 70% of respondents reported no prior experience with AsyncAPI. This unfamiliarity may have influenced their initial impressions of the tools and methods being evaluated. Participants were also asked about their familiarity with key concepts such as asynchronous communication, YAML, JSON, Ecore modeling tools, and graphical editors. Responses indicated varying levels of familiarity, with higher familiarity generally reported for JSON and graphical editors, while knowledge of Ecore modeling tools was less widespread.

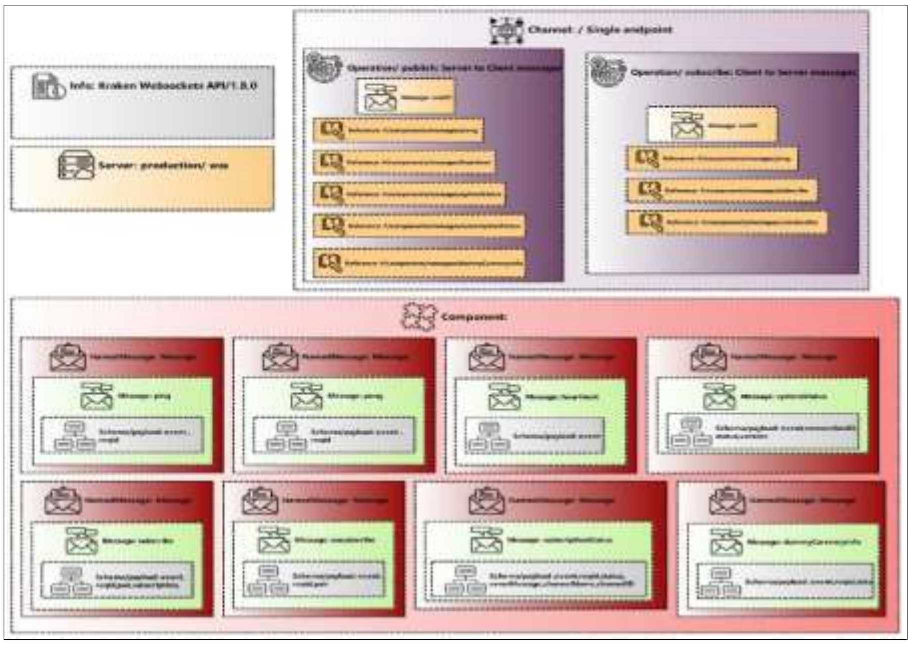

Channel: / Single endpoint
Info: Kraken Websockets API/1.0.0
Server: production/ wss
Operation/ publish: Server to Client message
Operation/ subscribe: Client to Server messages
Component

Figure 5. The graphical model for Case 1

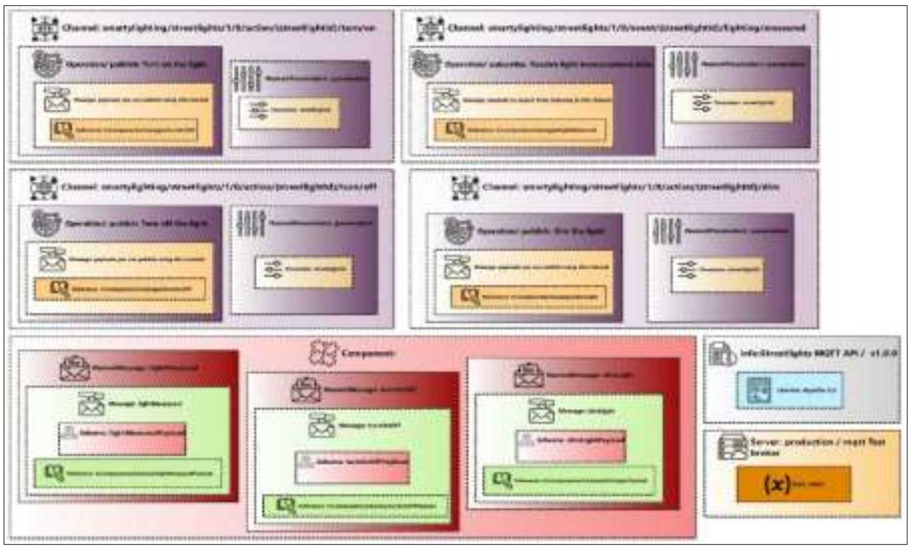


Figure 6. The graphical model for Case 2

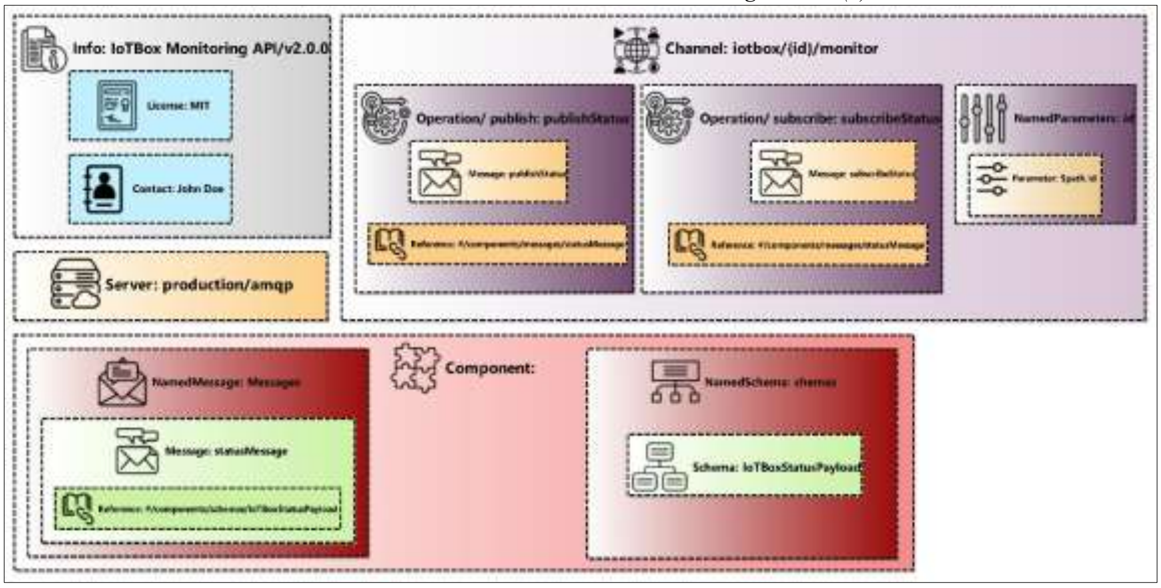


Figure 7. The graphical model for Case 4

Regarding the comprehension of different modeling methods, participants rated their understanding of the graphical model significantly higher compared to the tree model or code. Specifically, 60% of participants agreed or strongly agreed that they found the graphical model easy to understand. Additionally, participants estimated that the time required to understand the graphical model was shorter compared to other methods, suggesting a lower learning curve for graphical modeling.

In terms of efficiency, 62.5% of respondents believed that graphical models were faster to create than tree-based models or generated code. When asked which representation they understood the quickest, 67.5% selected the graphical model, further emphasizing its user-friendly nature. Similarly, the graphical model was considered less error-prone, with 62.5% of participants rating it as the most reliable method compared to tree-based or code-based approaches. Participants also evaluated the ease of use of various interaction methods. Drag-and-drop interfaces were overwhelmingly preferred, with 75% finding them easier than point-and-click or manual coding. Furthermore, the graphical editor emerged as the most stable and suitable environment, receiving 58.8% of the votes for stability and 47.5% as the preferred method for designing real AsyncAPI projects. A hybrid approach, combining multiple methods, was also favored by 37.5%, indicating that some participants saw value in blending different modeling approaches.

When asked about desired changes to the graphical model editor, the most common suggestions were adding new features (35%) and reducing complexity (32.5%). These recommendations highlight areas for improvement and reflect the participants' desire for a more comprehensive yet user-friendly tool.

Overall, the results suggest that graphical modeling is perceived as a faster, more intuitive, and less error-prone method for AsyncAPI projects, especially for users who may be less familiar with the domain. However, the diverse responses and preferences for hybrid approaches also underscore the need for flexible tools that can accommodate varying user needs and expertise levels.

Table 5. The results of the questionnaire

Q#	Answers
Q1	Bachelor (12.5%), Master Student (30.0%), Master (25.0%), PhD Student (12.5%), PhD (20.0%)
Q2	No (70%), Yes (30)
Q3	Very low (15%), Low (20%), Moderate (45%), High (15%), Very high (5%)
Q4	Very low (40%), Low (27.5%), Moderate (27.5%), High (0%), Very high (5%)
Q5	Very low (7.5%), Low (12.5%), Moderate (37.5%), High (25%), Very high (17.5%)
Q6	Very low (17.5%), Low (10%), Moderate (22.5%), High (27.5%), Very high (22.5%)
Q7	Very low (2.5%), Low (7.5%), Moderate (40%), High (32.5%), Very high (17.5%)
Q8	Very low (2.5%), Low (15%), Moderate (27.5%), High (37.5%), Very high (17.5%)
Q9	Very low (2.5%), Low (12.5%), Moderate (25%), High (45%), Very high (15%)
Q10	Very low (2.5%), Low (10%), Moderate (27.5%), High (22.5%), Very high (37.5%)
Q11	Very low (35%), Low (0%), Moderate (57.5%), High (0%), Very high (7.5%)
Q12	Very low (22.5%), Low (0%), Moderate (52.5%), High (0%), Very high (25%)
Q13	Very low (10%), Low (0%), Moderate (35%), High (0%), Very high (55%)
Q14	No (12.5%), Not sure (25%), Yes (62.5%)
Q15	Code (20.0%), Tree Model (12.5%), Graphical Model (67.5%)
Q16	Code (20.0%), Tree Model (17.5%), Graphical Model (62.5%)
Q17	Programming (2.5%), Point-and-click (17.5%), Drag and drop (75.0%)
Q18	Programming Editor (24.7%), Tree Editor (26.5%), Graphical Editor (58.8%)
Q19	Very low (0%), Low (0%), Moderate (20%), High (57.5%), Very high (22.5%)
Q20	No changes (15.0%), Increasing clarity (17.5%), Reducing complexity (32.5%), Adding new features (35.0%)
Q21	Programming Editor (12.5%), Tree Editor (2.5%), Graphical Editor (47.5%), Hybrid (37.5%)

An open-ended question revealed participant preferences for AsyncAPI modeling. Those favoring the graphical editor cited its ability to streamline complex processes, reduce cognitive load, and visually group components. Some preferred tree-based models for simpler projects due to their straightforward nature. YAML proponents typically had prior experience with the format but acknowledged its steep learning curve for newcomers.

Participant feedback on the graphical editor highlighted its intuitive interface, with drag-and-drop and point-and-click functionality simplifying model creation and minimizing errors. The simultaneous generation of tree-based and graphical models was also appreciated for catering to diverse user preferences. Real-time validations, like consistency checks, were praised for improving model accuracy by enabling quick error identification and correction.

Participants expressed concerns about the editor's reliance on Eclipse, and finding installation and configuration complex, especially for those unfamiliar with the environment. They suggested web-based or cloud-native solutions for better accessibility. Additionally, the lack of features like zooming, filtering, and collapsible sections made working with large or complex models difficult, and participants recommended adding these functionalities to reduce visual clutter and improve usability.

7. Discussion

Overall, the evaluation's findings demonstrated the advantages and disadvantages of the suggested AsyncAPI graphical editor while providing concise answers to the stated research questions. Remarkably, the 40-person sample size is small and might not be representative of the general population. Although histograms are excellent for visualizing distributions, they do not reveal more complicated relationships between variables. Please refer to the following correlation analysis (Figure 8) for more information and correlation of these relationships. Nevertheless, the participants' experience with the different AsyncAPI modeling techniques is hardly touched by this high-level fast.

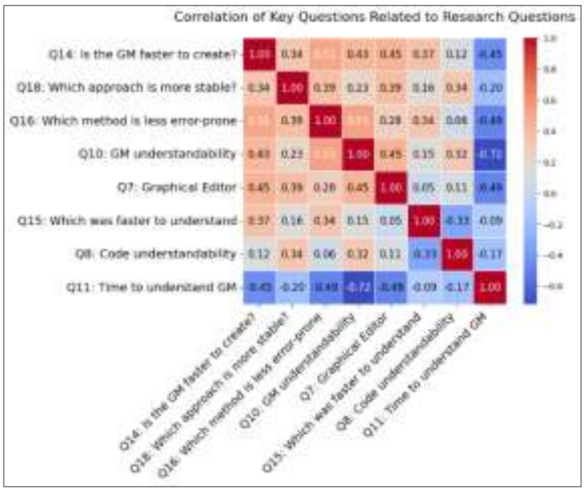


Figure 8. The heatmap diagram of correlation for key questions

RQ1: Is the graphical editor more usable, and does it lower the learning curve for developers, compared to tree-based representations and YAML?

Based on the correlation matrix analysis, we can conclude that the graphical editor is slightly more usable than at least some other representations due to a lower learning curve. Positive correlations were found between "Is the GM faster to create?", "GM understandability" (Q10), "GM understandability" (Q14), and the general sentiment towards the "Graphical Editor" (Q7) indicating that participants found the graphical editor faster and easier to understand. Furthermore, the positive correlation between Q15 and Q14, as well as Q10, supports the claim that the graphical editor improved understanding of the data. On the other hand, because Q11 (Time to understand GM) is negatively correlated with Q10 and Q7, it can be concluded with some certainty that less time was required to comprehend the graphical editor's functions and capabilities, indicating a lower learning curve. Additionally, Q8 (code understandability) had a higher correlation with Q10 than with Q15, implying that understanding code was more difficult than understanding its graphical representation. Together, this evidence points to a more user-friendly experience enabled by the graphical editor, which may reduce the learning curve for developers.

RQ2: Does the graphical editor reduce errors compared to both tree-based and YAML approaches for AsyncAPI modeling?

The correlation matrix analysis suggests that the graphical editor is perceived as less error-prone compared to other approaches. The positive correlation between Q16 (Which method is less error-prone?) and both Q10 (GM understandability) and Q14 (Is the GM faster to create?) indicates that participants associated the graphical editor with fewer errors and faster creation times. This association is further supported by the negative correlation between Q16 and Q11 (Time to understand GM), implying that the ease of understanding the graphical editor may contribute to a reduction in errors. The combination of these correlations strengthens the argument that the graphical editor effectively minimizes errors in AsyncAPI modeling.

RQ3: How do various representations, including graphical, tree-based, and YAML-based formats, compare in terms of efficiency and user experience in AsyncAPI modeling?

We can comparatively analyze the interrelationship of different questions to discover the relative efficiency and user experience for different representations (graph-based, tree-based, & YAML-based). A clear comparison can be found by exploring the correlations between the questions associated with each representation. In this sense, the comparison of Q10 (GM understandability) to Q8 (Code understandability) indicates which representation was easier to understand. Similarly, comparing Q14 (Is the GM faster to create?) with Q18 (Which approach is more stable?) and Q16 (Which method is less error-prone?) provides a holistic view of the efficiency and user experience associated with each approach. For example, a high positive correlation between Q14, Q10, and Q16, coupled with a high negative correlation between Q11 and these questions, would strongly suggest that the graphical editor provides a superior user experience in terms of creation speed, ease of understanding, and error reduction. Inter-correlations among all relevant questions provide a means to achieving this holistic view.

Implications and Limitations

The standardization of graphical representations for AsyncAPI specifications is significantly impacted by these findings. As a bridge between domain experts and technical experts, the graphical editor can encourage broader AsyncAPI adoption among heterogeneous teams. Compatibility is ensured by its smooth integration with current workflows, which reduces the need for significant retraining or modifications to current toolkits. Scalability, however, is a problem when simulating big, intricate systems.

Clarity may be hampered by visual clutter as the number of components and connections rises. This could be fixed in the future by implementing sophisticated visualization strategies like dynamic zooming, hierarchical abstractions, and selective highlighting. Additionally, less experienced users faced a steeper learning curve, whereas experienced modelers found the editor intuitive. Features like context-sensitive instructions and interactive tutorials could greatly increase accessibility for more individuals.

Lastly, adding real-time collaboration capabilities and interfaces that adapt to ranging user levels of proficiency could improve the editor's usability even more. To confirm the tool's effectiveness and determine additional areas for development, future studies should concentrate on testing it with a variety of user groups in real-world scenarios.

8. Conclusion

In this paper, we studied, designed, and evaluated a graphical editor specifically designed for AsyncAPI modeling, considering its feasibility and usability. We qualitatively showed improvements in the AsyncAPI specification comprehensibility of the proposed editor through graphical and tree-based representations and collected theoretical foundations for more complex systems. By providing a visual and interactive interface that helps reduce error rates and a higher level of abstraction, which improves the modeling experience for most users already familiar with graphical tools, it fulfills some important shortcomings of previous approaches.

The findings confirmed that using graphical editing helped users better understand the situations in which they were working. All participants agreed that the graphical representation facilitated faster interpretation of system structures and relationships. In nature, such support for graphical syntax demonstrates the second position, and it not only matches the research query of graphical syntax encouragements, but it also provides solid proof that this process is far superior to other alternatives such as tree-based models or textual representation.

One of the critical contributions of this editor is its seamless integration into existing modeling environments. By supporting standard modeling practices while introducing specific enhancements for AsyncAPI, the editor demonstrates practical value for novice and experienced modelers. Its ability to support rapid prototyping and error detection makes it a valuable tool for streamlining the development of messaging architectures in various domains, such as microservices, event-driven systems, and IoT applications.

While our evaluation demonstrates the effectiveness of the graphical editor for a range of case studies, further research is needed to assess its scalability with significantly larger and more complex AsyncAPI specifications. Performance and usability could potentially be affected by the sheer volume of elements in very large models, requiring further optimization and testing. The editor's current reliance on the Eclipse platform presents a barrier to entry for users unfamiliar with this environment. The installation and configuration process can be challenging for those without prior Eclipse experience. This contrasts with the trend towards web-based and cloud-native solutions, which offer greater accessibility and align with modern software development practices. Future work will explore migrating the editor to a more accessible platform to broaden its user base. As noted by participants in our evaluation, navigating and manipulating highly complex AsyncAPI models within the editor can be challenging. The current version lacks advanced features such as zooming, filtering, and collapsible sections, which would significantly improve the user experience when working with large models. These features are planned for future development to enhance the editor's usability for complex use cases.

In conclusion, the graphical editor represents a significant step forward in AsyncAPI modeling, bridging the gap between technical and domain-specific perspectives. Its intuitive interface and practical capabilities demonstrate the potential of graphical syntax to transform how developers and system architects design and implement messaging architectures. By addressing the identified limitations and exploring new avenues for innovation, this approach can continue to evolve as a foundational tool for modern software development practices.

9. References

[1] Nowick, S. M., & Singh, M. (2015). Asynchronous design-part 1: Overview and recent advances. IEEE Design and Test, 32(3), 5–18. doi:10.1109/MDAT.2015.2413759.

[2] Silva, J. (2024). AsyncAPI-First Design for Event-Driven Architectures: Improve Developer Experience. Master Thesis, University of Porto, Porto, Portugal. (In Portuguese).

[3] Oriol, M., Gómez, A., & Cabot, J. (2024). AsyncSLA: Towards a Service Level Agreement for Asynchronous Services. Proceedings of the ACM Symposium on Applied Computing, 1781–1788. doi:10.1145/3605098.3636074.

[4] Gómez, A., Iglesias-Urkia, M., Belategi, L., Mendialdua, X., & Cabot, J. (2022). Model-driven development of asynchronous message-driven architectures with AsyncAPI.

Software and Systems Modeling, 21(4), 1583–1611. doi:10.1007/s10270-021-00945-3.

[5] Rabii, A., Assoul, S., & Roudiès, O. (2022). Guide to domain specific language graphical editor prototyping. Computer Assisted Methods in Engineering and Science, 28(3), 243-261.

[6] Zhou, S., Sun, J., Xu, K., & Wang, G. (2024). AI-Driven Data Processing and Decision Optimization in IoT through Edge Computing and Cloud Architecture. Journal of AI-Powered Medical Innovations (International Online ISSN 3078-1930), 2(1), 64–92. doi:10.60087/vol2iisue1.p006.

[7] Pedro, B. (2024). Building an API Product: Design, implement, release, and maintain API products that meet user needs. Packt Publishing Ltd, Birmingham, United Kingdom .

[8] Lercher, A., Glock, J., Macho, C., & Pinzger, M. (2024). Microservice API Evolution in Practice: A Study on Strategies and Challenges. Journal of Systems and Software, 215. doi:10.1016/j.jss.2024.112110.

[9] Tzavaras, A., Mainas, N., & Petrakis, E. G. M. (2023). OpenAPI framework for the Web of Things. Internet of Things (Netherlands), 21. doi:10.1016/j.iot.2022.100675.

[10] Zafar, A., Azam, F., Latif, A., Anwar, M. W., & Safdar, A. (2024). Exploring the Effectiveness and Trends of Domain-Specific Model Driven Engineering: A Systematic Literature Review (SLR). IEEE Access, 12, 86809–86830. doi:10.1109/ACCESS.2024.3414503.

[11] Ternes, B., Rosenthala, K., & Streckera, S. (2021). User Interface Design Research for Modeling Tools: A Literature Study. Enterprise Modelling and Information Systems Architectures, 16(4), 1–30. doi:10.18417/emisa.16.4.

[12] Ray, P. P. (2017). A Survey on Visual Programming Languages in Internet of Things. Scientific Programming, 2017(1). doi:10.1155/2017/1231430.

[13] Ordoñez, K., Hilera, J., & Cueva, S. (2022). Model-driven development of accessible software: a systematic literature review. Universal Access in the Information Society, 21(1), 295–324. doi:10.1007/s10209-020-00751-6.

[14] Arslan, S., Kardas, G., & Alfraihi, H. (2024). On the Usability of a Modeling Language for IoT-Based Public Transportation Systems. Applied Sciences (Switzerland), 14(13). doi:10.3390/app14135619.

[15] David, I., Latifaj, M., Pietron, J., Zhang, W., Ciccozzi, F., Malavolta, I., Raschke, A., Steghöfer, J. P., & Hebig, R. (2023). Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. Software and Systems Modeling, 22(1), 415–447. doi:10.1007/s10270-022-01010-3.

[16] Wang, H., Wang, G., Gao, J., Hu, J., Li, J., & Zhang, H. (2023). Enhancing IoT Service Interface Through AsyncAPI with Extensions. Communications in Computer and Information Science. Springer, Singapore. doi:10.1007/978-981-99-4402-6_26.

[17] Verbruggen, C., & Snoeck, M. (2023). Practitioners' experiences with model-driven engineering: a meta-review. Software and Systems Modeling, 22(1), 111–129. doi:10.1007/s10270-022-01020-1.

[18] Bedini, F., Maschotta, R., & Zimmermann, A. (2021). A generative Approach for creating Eclipse Sirius Editors for generic Systems. 2021 IEEE International Systems Conference (SysCon), 1–8. doi:10.1109/syscon48628.2021.9447062.

[19] Budinsky, F. (2004). Eclipse modeling framework: a developer's guide. Addison-Wesley Professional, Boston, United States.